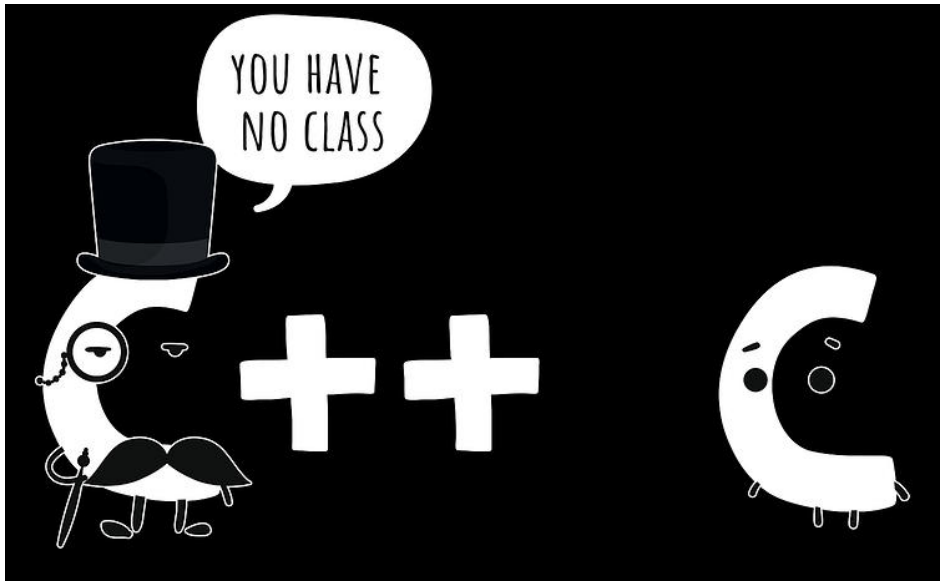
# The Hardware/Software Interface

## Structs and Alignment

**Instructors:**
Justin Hsia, Amber Hu

**Teaching Assistants:**
Anthony Mangus
Grace Zhou
Jiuyang Lyu
Kurt Gu
Mendel Carroll
Naama Amiel
Rose Maresh
Violet Monserate
Divya Ramu
Jessie Sun
Kanishka Singh
Liander Rainbolt
Ming Yan
Pollux Chen
Soham Bhosale




https://pixels.com/featured/1-computer-programmer-funny-c-class-joke-noirty-designs.html

# Relevant Course Information

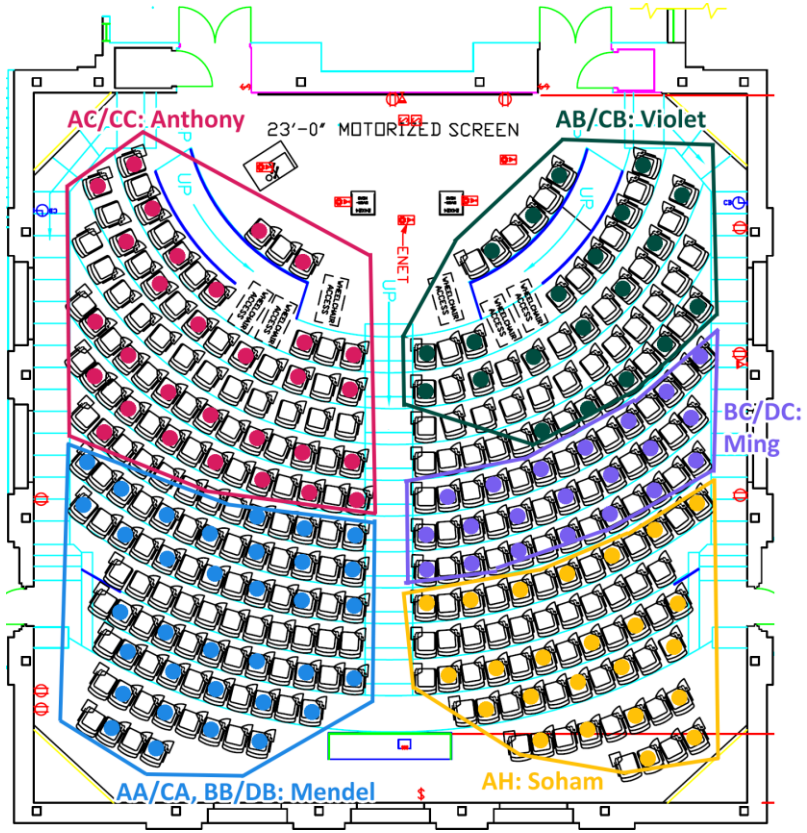
- Lab 2 due tonight
- Lab 3 released next Wednesday (10/29)
  - A shorter lab, due Friday, 11/7
- HW13 due next Wednesday (10/29), HW14 due next Friday (10/31)

- **Midterm** (Monday 10/27, 5:30-6:40 pm)
  - Midterm details Ed post #296
    - Come early to get exam and settle in
    - Make a cheat sheet! – two-sided letter page, *handwritten*
  - Review session (Ed post #301) tonight at 4:30 pm in CSE2 G01 and on Zoom

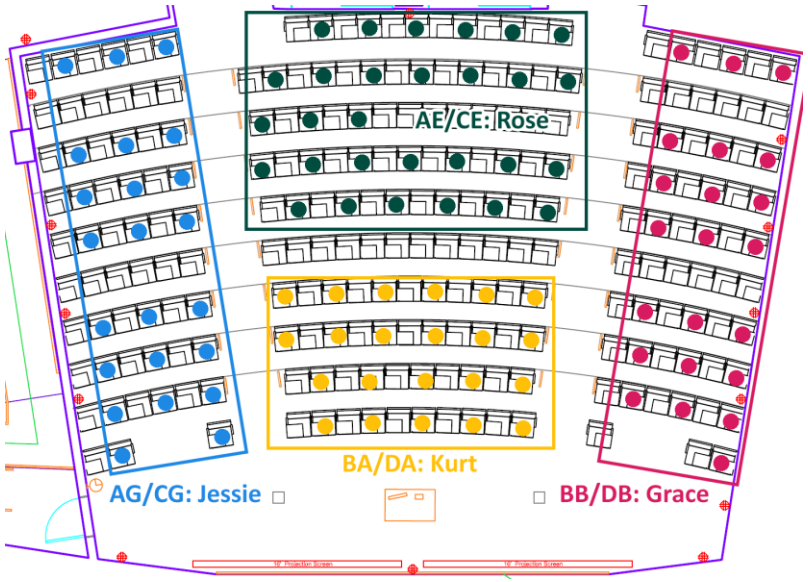
# Midterm Rooms and Sections

- Show up to space designated by your *registered* section!

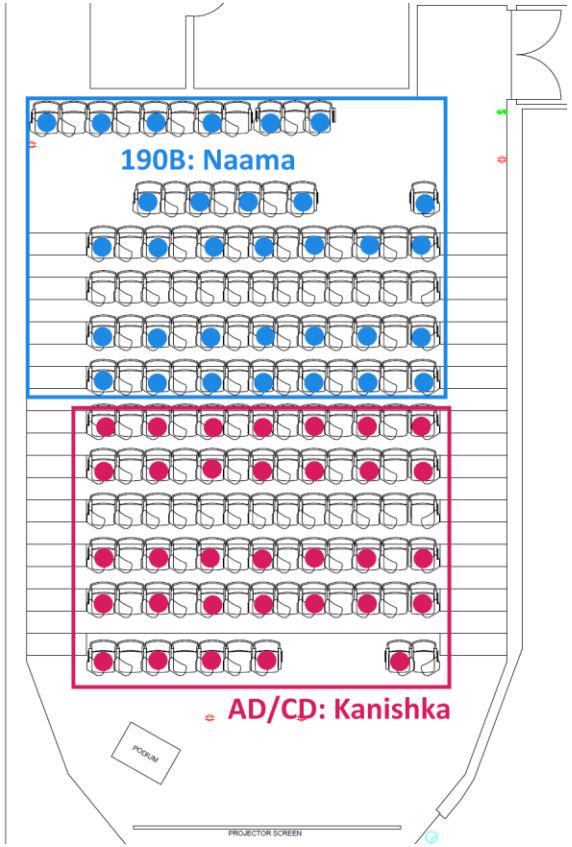
## ARC 147




## CSE2 G20




## SIG 134



# Lecture Outline (1/2)

- **Structs and Typedef in C**
- Struct Layout and Alignment

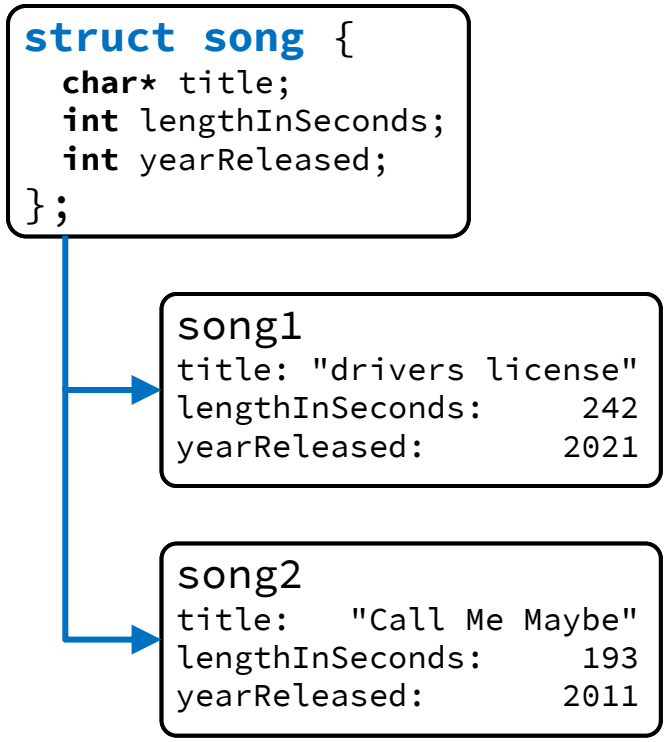
# Structs in C (Review)

- A structured group of variables, possibly including other structs
  - Way of defining compound data types

```
struct song {
  char* title;
  int lengthInSeconds;
  int yearReleased;
};

struct song song1;
song1.title = "drivers license";
song1.lengthInSeconds = 242;
song1.yearReleased = 2021;

struct song song2;
song2.title = "Call Me Maybe";
song2.lengthInSeconds = 193;
song2.yearReleased = 2011;
```



# Struct Definitions (Review)

- Structure definition:
  - Does NOT declare any instances
  - Creates new data type "**`struct struct_tag`**"

```
struct struct_tag {
  type_1 field_1;
  …
  type_N field_N;
};
```

Easy to forget semicolon!

- Variable declarations like any other data type:

```
struct struct_tag tag1, *pt, tag_ar[3];
```

instance pointer array

- Can also combine struct and instance definitions:
  - This syntax can be difficult to read, though:

```
struct struct_tag {
  /* fields */
} st, *p = &st;
```

# Scope of Struct Definition (Review)

- Why is the placement of struct definition important?
  - Compiler needs to know about this new symbol/data type
  - Without definition, program doesn't know how much space to allocate

```
struct data {
    int ar[4];
    long d;
};
```

← Size = **24** bytes

Size = **32** bytes →

```
struct rec {
    int a[4];
    long i;
    struct rec* next;
};
```

- Almost always define structs in global scope near the top of your file
  - Struct definitions follow normal rules of scope

# Typedef in C (Review)

- A way to create an *alias* for another data type:
  `typedef <data type> <alias>;`
  - After typedef, the alias can be used interchangeably with the original data type
  - *e.g.*, `typedef unsigned long int uli;`

- Joint struct definition and typedef:

```
struct struct_tag {
  /* fields */
};
typedef struct struct_tag alias;
alias a1;
```

```
typedef struct {
  /* fields */
} alias;
alias n1;
```

# Accessing Struct Fields (Review)

- Given a struct instance, access fields using the `.` operator:
  - ```
    struct rec r1;
    r1.i = val;
    ```
- Given a *pointer* to a struct:
  - ```
    struct rec* r = &r1;
    r->i = val;  // or (*r).i = val;
    ```

```
struct rec {
    int a[4];
    long i;
    struct rec* next;
} r1, *r = &r1;
```

- **In assembly:**
  - Holds address of the first byte in register (Rb) or label
  - Access fields with offsets (D)

```
# local struct
movslq  %edi, %rdi
movq    %rdi, -24(%rsp)
```

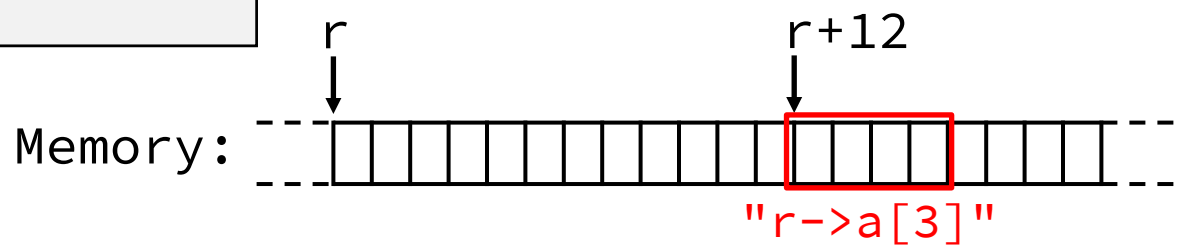
```
# global struct r1
movslq  %edi, %rdi
movq    %rdi, r1+16(%rip)
```

# Struct Pointers

- Pointers store addresses, which all "look" the same
  - Lab 0 Example: struct instance `Scores` could be treated as array of `int`s of size 4 via pointer casting
  - A struct pointer doesn't *have* to point to a declared instance of that struct type
- Different struct fields may or may not be meaningful, depending on what the pointer points to
  - This will be important for Lab 5!

```
int get_a3(struct rec* r) {
  return r->a[3];
}
```

```
  movl  12(%rdi), %eax
  ret
```



# Polling Questions (1/2)

```
struct ll_node {
  long data;
  struct ll_node* next;
} n1, n2;
```

- How much space does (in bytes) does an instance of `struct ll_node` take?

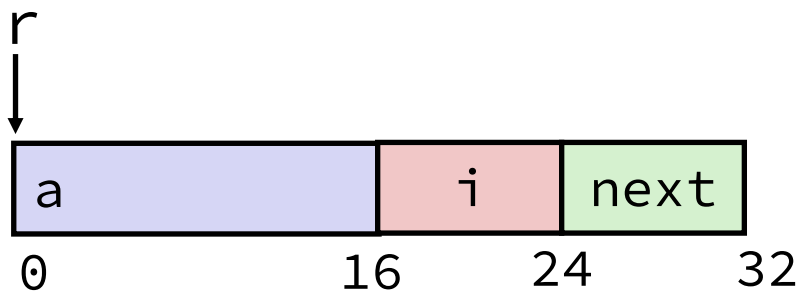
- Which of the following statements are syntactically valid?
  - `n1.next = &n2;`
  - `n2->data = 351;`
  - `n1.next->data = 333;`
  - `(&n2)->next->next.data = 451;`

# Lecture Outline (2/2)

- Structs and Typedef in C
- **Struct Layout and Alignment**

# Structure Representation (Review)

```
struct rec {
    int a[4];
    long i;
    struct rec* next;
} st, *r = &st;
```




- Structure represented as block of memory
  - Each instance is a contiguously-allocated region of memory big enough to hold all of the fields
- Compiler determines overall size + positions of fields
  - Fields ordered according to declaration order, even if another ordering would be more compact
  - Machine-level program has no understanding of the structures in the source code

# Alignment Principles (Review)

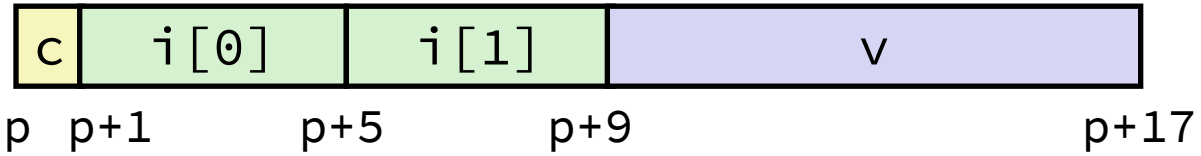
- Primitive data type of size $K$ bytes is considered *aligned* if its address is **a multiple of $K$**
  - Required on some machines; x86-64 hardware will work regardless
  - Memory accessed by aligned chunks of bytes (see: caching and virtual memory)
    - Inefficient to load or store value that crosses boundaries

- Address bit interpretation of alignment:

| $K$ | Type | Addresses |
|---|---|---|
| 1 | `char` | No restrictions |
| 2 | `short` | Lowest bit must be zero: $\ldots0_2$ |
| 4 | `int, float` | Lowest 2 bits zero: $\ldots00_2$ |
| 8 | `long, double, *` | Lowest 3 bits zero: $\ldots000_2$ |
| 16 | `long double` | Lowest 4 bits zero: $\ldots0000_2$ |

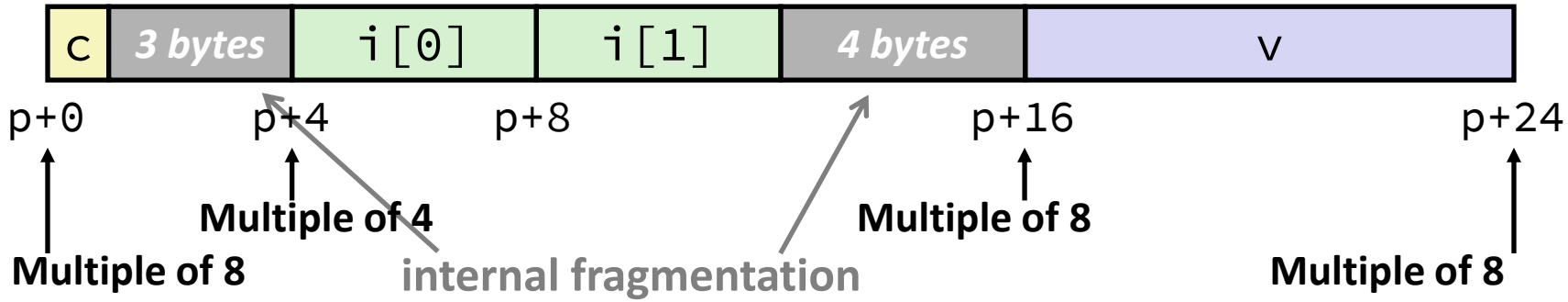
# Structures & Alignment (Review)

- Unaligned Data




```
struct S1 {
  char c;
  int i[2];
  double v;
} st, *p = &st;
```

- Aligned Data
  - Primitive data type of size $K$ bytes is considered *aligned* if its address is **a multiple of $K$**

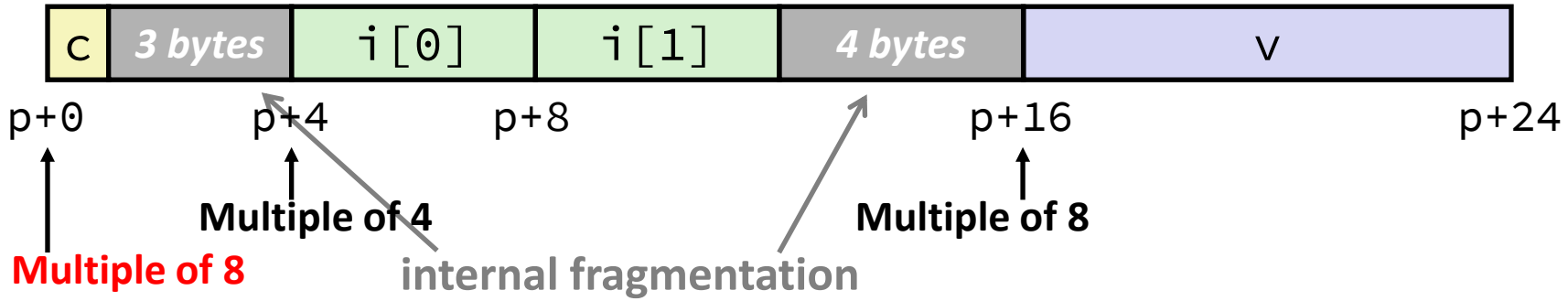


# Satisfying Alignment with Structures (1)

- Within struct we must satisfy each element's alignment requirement
- Overall structure placement has alignment requirement $K_{\max}$
  - $K_{\max}$ = largest alignment of any field

```
struct S1 {
  char c;
  int i[2];
  double v;
} st, *p = &st;
```

- Address of the struct must be a multiple of $K_{\max}$

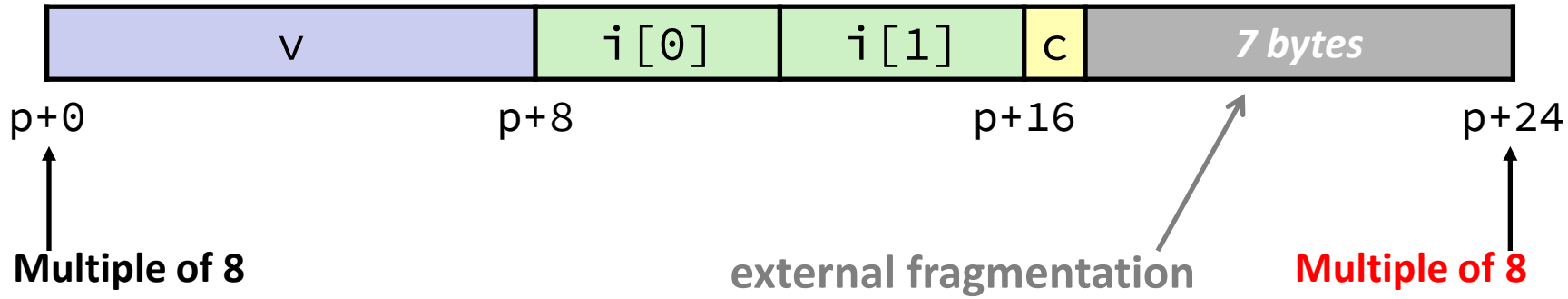


# Satisfying Alignment with Structures (2)

- Within struct we must satisfy each element's alignment requirement
- Overall structure placement has alignment requirement $K_{\max}$
  - $K_{\max}$ = largest alignment of any field

```
struct S2 {
  double v;
  int i[2];
  char c;
} st, *p = &st;
```

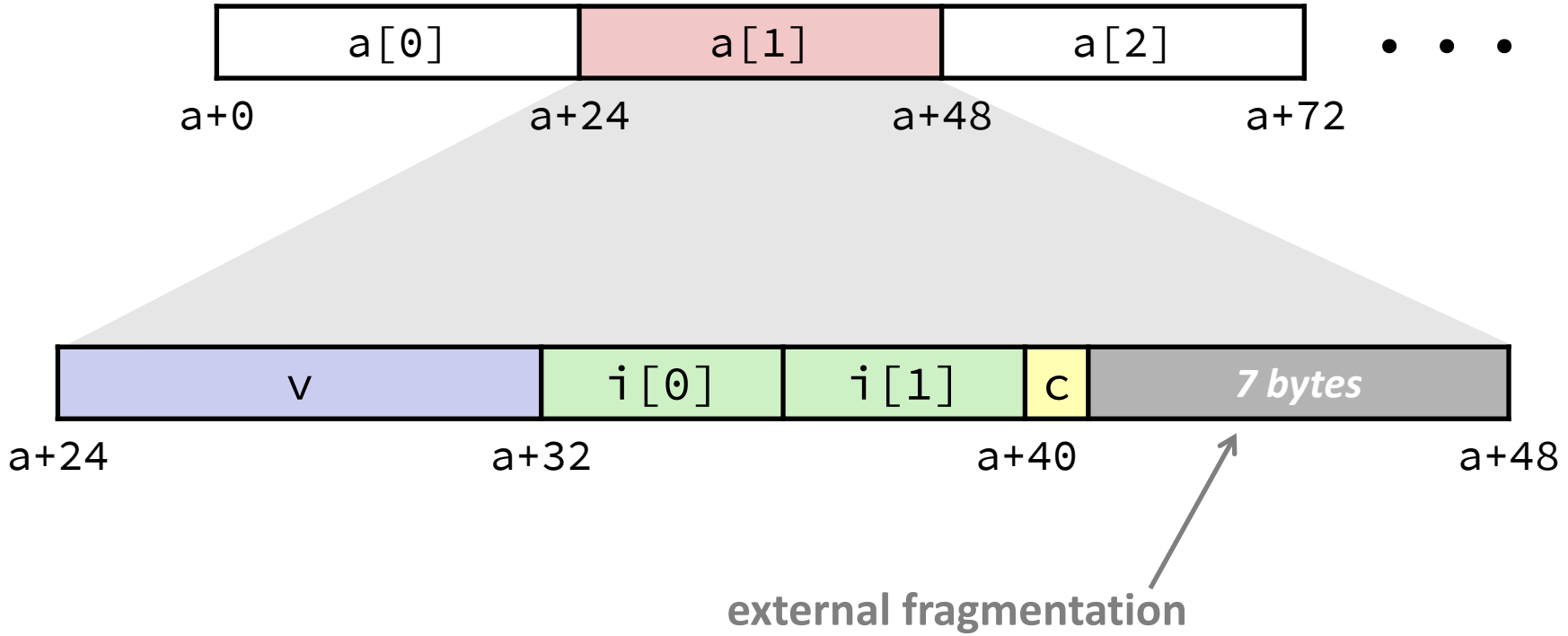
- Overall structure size must be multiple of $K_{\max}$



# Arrays of Structures

- Overall structure length multiple of $K_{max}$
- Satisfy alignment requirement for every element in array

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



# Alignment of Structs Summary

- Compiler will do the following:
  1) Maintains declared ordering of fields in struct
  2) Each ***field*** must be aligned within the struct to its $K$ *(may insert padding)*
  3) Overall struct ***size*** must be multiple of $K_{max}$ *(may insert padding)*
  4) When allocated, struct addresses will be ***aligned*** to $K_{max}$

- C Notes:
  - `sizeof()` can be used to get the overall size of structs
  - `offsetof()` can be used to get field offsets within structs

# Struct Layout Example (Review)

K
1
4
8
2

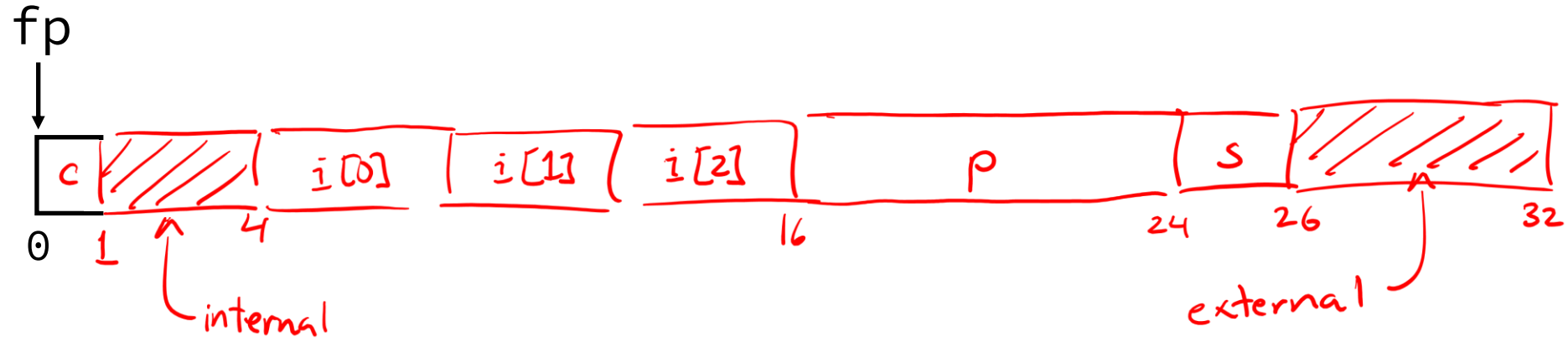
```
struct frag {
  char  c;
  int   i[3];
  struct frag* p;
  short s;
} f, *fp = &f;
```

$K_{max} = 8$

struct size = 32 B

internal fragmentation = 3 B

external fragmentation = 6 B

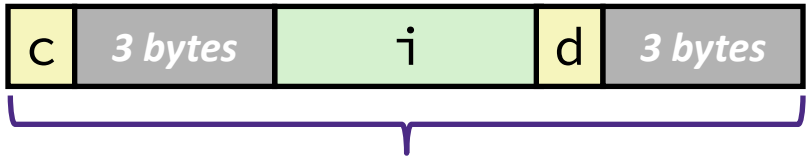


# How the Programmer Can Save Space

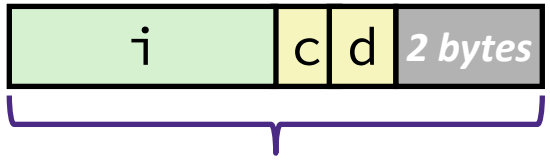
- Compiler must respect order elements are declared in
  - Rule of thumb: declare field in decreasing order of alignment requirement

```
struct S3 {
  char c;
  int i;
  char d;
} st;
```

```
struct S4 {
  int i;
  char c;
  char d;
} st;
```




12 bytes

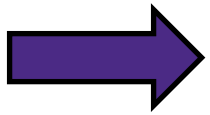



8 bytes

# Polling Questions (2/2)

- Minimize the size of the struct by re-ordering the fields:

```
struct old {
  int i;
  short s[3];
  char* c;
  float f;
};
```



```
struct new {
  int    i;
  _______ _______;
  _______ _______;
  _______ _______;
};
```

  - What is the minimum size of `struct new`?

    A. 22 bytes  B. 24 bytes

    C. 28 bytes  D. 32 bytes

# Homework Setup (if time)

- Struct in a struct?
  - It's just another data type, with its own alignment requirement
  - Example:

```
struct outer {
  char c;
  struct inner {
    short s;
    int i;
  } in;
};
```

→

```
struct inner {
  short s;
  int i;
};
struct outer {
  char c;
  struct inner in;
};
```

# Summary

- Alignment
  - Data of alignment requirement (*i.e.*, size) $K$ is considered aligned if its address is a multiple of $K$
  - Arrays have alignment requirement of an individual element, not the total size

- Structures
  - Allocate bytes for fields in order declared by programmer – can make choices to minimize memory allocations
  - Pad in middle to satisfy individual element alignment requirements ($K$)
    - ***Internal* fragmentation**
  - Pad at end to satisfy overall struct alignment requirement ($K_{max}$)
    - ***External* fragmentation**